CEWES MSRC/PET TR/99-13

The Benefits of Fortran Procedure Interfaces

by

Dan Nagle

04h01299

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

The Benefits of Fortran Procedure Interfaces

Dan Nagle

One of the strengths of modern, high-level programming languages is the ability to compile subprograms of a larger program independently. Independent compilation gives the ability

- to supply different compiler options for different subprograms;
- to compile a procedure once and use it in more than one program or in a library;
- to re-compile only those subprograms that have been modified without rebuilding the entire program.

The price Fortran programmers have paid for these conveniences is that compilers have not been able to do much in the way of interprocedural analysis. Thus, a major source of errors in Fortran programs has been misuse of subprograms. Typical bugs include calling a subprogram with incorrect argument types or with the wrong number of arguments.

Fortran 90 introduces to Fortran some new features making subprograms less error-prone and more robust. Use of these features requires that the calling subprogram contain information about the called subprogram. Such information is supplied by a new language construct called an *interface block*, similar to a C function prototype. The interface block defines the procedure's interface, which includes whether the procedure is a function or a subroutine. If the procedure is a function, the type and attributes of the function result must be specified. In either case, the types, positions, attributes, and names of dummy arguments are specified in an interface block.

All procedures in Fortran have an interface, which may be explicit or implicit. An implicit interface is obtained from the usage of the procedure. Basically, the compiler assumes that the given usage is correct. An explicit interface, on the other hand, states what the interface should be, so the compiler can check the procedure's usage. All intrinsic procedures have an explicit interface. An explicit interface to a user-defined procedure may come about in one of the following three ways:

1. The procedure is internal (i.e., its source code follows the CONTAINS statement within the calling subprogram).

2. The procedure is in a module.

3. The calling procedure has an interface block for the called subprogram.

Generally, a procedure interface may not be specified twice. Thus, a procedure interface must not be specified for an internal procedure or a module procedure, because the interface is already known, as described above. An interface block may be used to give a procedure an explicit interface, a generic interface, to use a procedure as a user-defined operator, or to use a procedure as a user-defined assignment.

The rest of this note discusses Fortran interface blocks, the advantages of using them (specifically,

the new features of procedure arguments), and how to use them. The simple interface block will be discussed, along with generic interfaces, user-defined operators, and user-defined assignment.

Interface Blocks

An interface block contains the declaration of the procedure, and the declarations of the procedure's dummy arguments, if any. The basic form of the interface block is as follows.

```
interface
!   procedure declarations go here
end interface
```

While the interface block is placed in the calling program in the declarative section, it is really telling about the called procedure. Therefore, interface blocks are their own scope. This is not a problem, since the interface block contains a description of the procedure and each of its arguments. The effect here is that an IMPLICIT statement in the subprogram containing the interface block has no effect inside the interface block.

As a specific example, consider a typical subroutine for matrix inversion, named `MINV()`, with a 2-dimensional array to be inverted in-place, the size of the array, the tolerance accepted for pivot elements, and returning the determinant and an error flag. Its interface block will look like the following (indenting optional):

```
interface
   subroutine minv( a, n, tol, det, ierr)
      real, dimension( n, n), intent( inout) :: a
      integer, intent( in) :: n
      real, intent( in) :: tol
      real, intent( out) :: det
      integer, intent( out) :: ierr
   end subroutine
end interface
```

With this interface block in scope, the compiler may check that usage of `minv()` is correct. For example, the erroneous call (assuming implicit typing)

```
  call minv( a, d, t, n, ierr)      ! WRONG! d & n are reversed!
```

will be caught at compile time (and not found later during debugging as the cause of the crash). There are also advantages for the programmer in terms of optimization. The compiler knows an argument's INTENT along with its other attributes. Thus, the compiler knows when a variable on an actual argument list will be modified and when one will not, and can do a better job of keeping data in register.

Additionally, many of the new features of Fortran require use of explicit interfaces. A list of

features requiring explicit interfaces follows.

- optional arguments
- keyword arguments
- a function returning an array
- a function returning a pointer
- a function returning a character string whose length will be computed during execution (i.e., a function returning character*(*))
- assumed shape array arguments
- pointer or target dummy arguments
- user-defined generic procedures
- user-defined operators
- user-defined assignment
- pure or elemental procedures.

An example of an explicit subroutine has already been shown. The interface of a recursive subroutine follows.

```
interface
   recursive subroutine findit( a)
      real, dimension( :, :), intent( inout) :: a
   end subroutine
end interface
```

Notice that the dimensions of A are ":" (A is an assumed-shape array), the array A may be passed without its extents. Because the compiler knows the calling sequence, an "array descriptor" will be passed rather than the address of the array. This means that the called routine can know the rank, extents, and upper and lower bounds of each dimension of A. This eases the programmer's burden because the compiler writes the "implementation detail" code.

PURE and ELEMENTAL subroutines may be declared by putting the PURE or ELEMENTAL keyword on the subroutine statement (assuming, of course, that it adheres to the restrictions for PURE or ELEMENTAL subroutines). Examples follow.

```
interface
   pure subroutine compute( a, b)
      real, dimension( :, :), intent( in) :: a
      real, dimension( :, :), intent( out) :: b
   end subroutine
end interface
```

The interested reader is referred to the CEWES MSRC PET Technical Report entitled *The Effect of Fortran 95 PURE and ELEMENTAL Procedures on Parallel Execution* for more information on the uses of PURE subroutines. Again, the array dimensions need not be passed.

```
interface
   elemental subroutine filter( a, b)
      real, intent( in) :: a
      real, intent( out) :: b
   end subroutine
end interface
```

The interested reader is again referred to the above mentioned report for more information on elemental subroutines. The elemental procedure may be applied to an array, but it is declared in a scalar manner.

There is more to specify in the interface to a function. This is because the function will return a value via the function result, which will also have type, kind and rank information specified. Here is a simple function interface.

```
interface
   integer function get_io_unit( minunit, maxunit)
      integer, intent( in) :: minunit
      integer, intent( in) :: maxunit
   end function
end interface
```

Here, the function return type is integer. Since neither argument will be modified, they may be declared INTENT(IN). The compiler may use the information to better optimize the code in both the caller and the callee.

A Fortran 90/95 function may return an array result. In order to do so, the function must have an interface to tell the compiler that the result will be an array. Here is an interface for an array-returning function.

```
interface
   real function array_fnctn( a, n)
      dimension array_fnctn( n, n)
      real, dimension( n, n) :: a
   end function
```

```
end interface
```

A pointer example follows.

```
interface
   real function ptr_fnctn( a, n)
      pointer ptr_fnctn
      real, dimension( n, n) :: a
   end function
end interface
```

The tired, old example of the factorial calculation illustrates a recursive function.

```
interface
   recursive integer function factorial( n) result( f)
      integer, intent( in) :: n
   end function
end interface
```

This interface block has the declarations of the arguments as before, but with the recursive keyword. What is new here is the `result()` clause. Why is there a `result()` clause? Remember that in Fortran, the function name may be used within the function as a local variable, with the last value of that variable being the function return value. With recursion, there is a problem. Is the use of the function name, on the right hand side of an equal sign, a reference to the local variable (which could be an array) or a new invocation of the function? The issue is resolved by using the result clause to name the local variable whose final value will be the function return value. This situation is demonstrated below.

```
recursive integer function factorial( n) result( f)
   integer, intent( in) :: n
   if( n == 0 )then
      f = 1
   else
      f = n * factorial( n-1)
   endif
end function
```

Procedure Arguments

This section discusses the use of some of the new features of subprogram arguments. Use of these features requires an explicit interface. For external subprograms, the explicit interface is made by an interface block.

An assumed shape array is a dummy argument array whose extents are declared to be ":". The advantage here is that the programmer need not pass the array's extents explicitly. As mentioned

above, use of assumed shape arrays requires an explicit interface so the compiler will know to pass an array descriptor rather than the array address.

A dummy argument's INTENT describes whether the actual argument will be read, or written, or both by the subprogram. It is to the programmer's advantage to specify an argument's INTENT because it provides more information to the compiler's optimizer, which can then write faster code. The compiler will check, when compiling the callee, that the INTENT is honored. Also, having an argument's INTENT specified helps the programmer read and understand the program.

If a subprogram has an explicit interface, it may be called with keyword arguments. The arguments may appear in any order, which is sorted out by the keyword. An example follows.

```
interface
   subroutine example( a, b, c)
      real, intent( out) :: a
      real, intent( in) :: b, c
   end subroutine
end interface
```

With this interface block in scope, `example()` may be called in any of the following ways:

```
   call example( x, y, z)
   call example( a=x, b=y, c=z)
   call example( c=z, a=x, b=y)
   call example( c=z, b=y, a=x)
```

In each case, the compiler has enough information to correctly map the actual arguments to the right dummy argument. The programmer is helped by having argument names (which may be meaningful) rather than argument positions to remember.

Fortran 90 has standardized the handling of optional arguments to subprograms. The Fortran 77 standard never allowed optional arguments, and the popular extension `NUMARG()` was error-prone as it never told the programmer *which* argument was missing. The following example modifies the earlier `MINV()` example to demonstrate the use of optional arguments, and the associated intrinsic logical function `PRESENT()`.

```
interface
   subroutine minv( a, n, tol, det, ierr)
      real, intent( inout),        &
      dimension( n, n)            :: a
      integer, intent( in)        :: n
      real, intent( in), optional :: tol
      real, intent( out)          :: det
      integer, intent( out)       :: ierr
   end subroutine
end interface
```

The above interface declares TOL to be an optional argument so MINV( ) may be called with or without TOL. In the example code below, the intrinsic PRESENT( ) is used to set a default tolerance if one is not specified in the argument list.

```
   if( present( tol) )then
      local_tol = tol


   else
      local_tol = epsilon( a)
   end if
```

In the above statement, the local variable LOCAL_TOL will be used as the tolerance. It will be set to TOL only if TOL is on the actual argument list. If TOL is not on the actual list, a default value (epsilon( ) of the kind of A) is used instead. Note that if some other actual argument were missing (say, ierr), it will be caught at compile time as a coding error.

If the programmer wishes to use a dummy argument which is a POINTER or TARGET, an explicit interface is required. For more information on Fortran pointers, please see the CEWES MSRC PET Technical Report entitled *Cray Fortran 90 Pointers v. Fortran 90 Pointers and Porting from the Cray C90 to the SGI Origin 2000*.

Generic Procedures

The idea of generic procedures was introduced in Fortran 77. The idea was to give one name to an intrinsic function, and let the compiler pick the correct specific function by examining the type of the argument. For example, the intrinsic square root function had different names depending on whether the argument was a real variable (SQRT( )), a double precision variable (DSQRT( )), or a complex variable (CSQRT( )). With generic intrinsic functions, one name (SQRT( )) would be used for the square root function, and the specific function used would be chosen by the compiler based upon the argument type. This feature made it possible to change a variable between real and double precision by simply changing the declaration of the variable. It was no longer necessary to change every intrinsic function where the variable appeared as an argument.

Now suppose the programmer wants to define a new function, a cube root function, and have the same generic name be used with variables of type real and complex (assuming that the processor supports kinds 4 and 8). First, the programmer will write the functions.

```
real function real4_cbrt( a)
   real( kind=4), intent( in) :: a
   real4_cbrt = a ** .33333333333333333_4
end function
real function real8_cbrt( a)
   real( kind=8), intent( in) :: a
```

```
   real8_cbrt = a ** .33333333333333333333_8
end function
complex function complex4_cbrt( a)
   complex( kind=4), intent( in) :: a
   complex4_cbrt = a ** .33333333333333333333_4
end function
complex function complex8_cbrt( a)
   complex( kind=8), intent( in) :: a
   complex8_cbrt = a ** .33333333333333333333_8
end function
```

Then the programmer writes the generic interface block.

```
interface cbrt
   real( kind=4) function real4_cbrt( a)
      real( kind=4), intent( in) :: a
   end function
   real( kind=8) function real8_cbrt( a)
      real( kind=8), intent( in) :: a
   end function
   complex( kind=4) function complex4_cbrt( a)
      complex( kind=4), intent( in) :: a
   end function
   complex( kind=8) function complex8_cbrt( a)
      complex( kind=8), intent( in) :: a
   end function
end interface
```

And now, finally, the programmer may use a generic name `cbrt()` for any variable of type real and complex.

Interfaces, Derived Types, and Extending Intrinsic Functions

With Fortran 90, the programmer may define new variable types, called derived types. In order for functions of these user-defined derived types to have a generic name, an interface block is used. Here is the derived type.

```
type :: velocity
   real :: v_x
   real :: v_y
   real :: v_z
end type
```

The programmer may now declare variables to be of type velocity. Suppose the programmer wants a square root function that may take a variable of type velocity as its argument. The function may be defined as follows:

```
type( velocity) function velocity_sqrt( x)
   type( velocity), intent( in) :: x
   velocity_sqrt%v_x = sqrt( x%v_x)
   velocity_sqrt%v_y = sqrt( x%v_y)
   velocity_sqrt%v_z = sqrt( x%v_z)
end function
```

In order to use this function with the generic name `sqrt()`, an interface block must be supplied as follows:

```
interface sqrt
   type( velocity) function velocity_sqrt( x)
      type( velocity), intent( in) :: x
   end function
end interface
```

Now a variable of type velocity may be passed to the `sqrt()` function, as shown in the following example.

```
type( velocity) :: a, b
   a = sqrt( b)
```

The general form of the interface block now has the generic name on the interface statement. This tells the compiler to use the function specified in the interface block to resolve calls to `sqrt()` when the argument is of type velocity. Thus, the programmer may extend an intrinsic function to apply to user-defined derived types.

Defining Operators

An interface block may also be used to define an operator. The programmer may extend an intrinsic operator or define a new one. To define a new operator or extend the definition of an existing one, the programmer must write a function of one or two arguments. The operator will then be either a unary operator or binary operator, respectively.

For example, consider the type velocity discussed in the previous examples. Since two velocities may be meaningfully added, one would like to have an operator which could add together two velocities. The following example shows how. First, define the velocity adding function. It must return a value of type velocity, and it must take exactly two velocities as arguments.

```
type( velocity) function velocity_add( v1, v2)
   type( velocity), intent( in) :: v1, v2
   velocity_add%v_x = v1%v_x + v2%v_x
   velocity_add%v_y = v1%v_y + v2%v_y
   velocity_add%v_z = v1%v_z + v2%v_z
end function
```

Next, define the interface block, as follows.

```
interface operator( +)
   type( velocity) function velocity_add( v1, v2)
      type( velocity), intent( in) :: v1, v2
   end function
end interface
```

And the programmer may now use the + operator between two velocities to compute a third, the sum, as follows.

```
type( velocity) :: u, v, w
u = v + w
```

The compiler will use the function `velocity_add` to compute the `v+w` sum. Note that by having the sum of velocities in a separate function, one may modify the function without disturbing the code using the `velocity_add()` function. For example, one could change the `velocity_add()` function to compute a relativistically correct addition of velocities without changing any code using it.

The programmer may invent new operators and use them in subprograms to simplify and make the code clearer. Recall the `MINV()`example. In order to use `MINV()` as an operator, it must be converted to a function returning an array. Also, every argument other than the array to be inverted must be eliminated. The following steps can be taken to eliminate the arguments:

- `N` can be eliminated by making A an assumed-shape array.
- `TOL` can be eliminated by using a default tolerance.
- `DET` can be eliminated by foregoing the determinant output.
- `IERR` can be eliminated by foregoing the error code.

With these changes, the interface to make `MINV()` an operator `.INVERSE.` is as follows:

```
interface operator( .inverse.)
   real function minv( a)
      dimension minv( size( a, dim= 1), size( a, dim=2) )
      real, dimension( :, :), intent( in) :: a
   end function
end interface
```

Now a square two-dimensional array may be defined, and the `.INVERSE.` operator can be applied to compute a new array of the same size.

```
real, dimension( 100, 100) :: a, b
b = .inverse. a
```

This version is easy to read, and is a more concise notation where details such as array size are handled automatically by the compiler, thereby reducing the opportunities for coding errors.

Defining Assignments

By default, standard Fortran can convert variables of some types into variables of another type. For example, a double precision value may be assigned to an integer. The programmer may wish to define other assignments (which are really conversions). Where the user-defined operator required a function, the user-defined assignment requires a subroutine of exactly two arguments, one INTENT(IN) and one INTENT(OUT).

A conversion from character to integer is implemented as follows. If the character is a digit, the integer value is the value of the digit (0-9). If the character is not a digit, the integer value will be negative as a flag.

```
subroutine char_to_int( c, i)
character, intent( in) :: c
integer, intent( out) :: i
   if( ichar( c) >= ichar( '0') .and. ichar( c) <= ichar( '9') )then
      i = ichar( c) - ichar( '0')
   else
      i = -1
   endif
end subroutine
```

This subroutine will do what is required (for ASCII characters). The following interface block can be used to define the assignment.

```
interface assignment( =)
   subroutine char_to_int( c, i)
      character, intent( in) :: c
      integer, intent( out) :: i
   end subroutine
end interface
```

Now, a character may be assigned directly to an integer, as follows:

```
character :: c = '4'
integer :: i
i = c
```

Note that one could not, at this stage, assign an integer to a character because that assignment has yet to be defined.

If a program were analyzing two-dimensional fluid flow, part of the calculation might be done

using complex variables. The programmer may also have defined a two-dimensional velocity type, similar to the three-dimensional type defined above. The program may require that data be transferred between the variables of the velocity type and variables of complex type. The programmer must write a conversion routine to accomplish this task. The following example shows the definition of the two dimensional velocity type.

```
type :: velocity2
   real :: vx
   real :: vy
end type
```

Now, here are the conversion routines from complex to velocity2, and vice versa.

```
   subroutine cmplx_to_v2( c, v)
      complex, intent( in) :: c
      type( velocity2), intent( out) :: v
         v = velocity2( real( c), aimag( c) )
   end subroutine
   subroutine v2_to_cmplx( v, c)
      type( velocity2), intent( in) :: v
      complex, intent( out) :: c
         c = cmplx( v%vx, v%vy)
   end subroutine
```

These subroutines may be called to perform the desired conversion. To have these routines called automatically by equating a complex variable to a `velocity2` variable, an interface block similar to the following must be used:

```
interface assignment( =)

   subroutine cmplx_to_v2( c, v)
      complex, intent( in) :: c
      type( velocity2), intent( out) :: v
   end subroutine

   subroutine v2_to_cmplx( v, c)
      type( velocity2), intent( in) :: v
      complex, intent( out) :: c
   end subroutine

end interface
```

With this interface in scope, the conversion routines may be invoked as any other assignment.

```
type( velocity2) :: v1, v2
complex :: c1, c2
```

```
v1 = c1
c2 = v2
```

The compiler will automatically call the correct conversion routine to perform the assignment.

Conclusion

Fortran interface blocks allow the compiler to do interprocedural analysis, resulting in more robust, error-free programs. Generic procedures allow an intrinsic function to have one name and let the compiler pick the correct specific function by examining the type of the argument. Operator overloading allows the programmer to extend an intrinsic operator or define a new one through the use of an interface block. With assignment overloading, not only can Fortran convert variables of some types into variables of other types, but the programmer may define other assignments as well. In addition, Fortran 90 has standardized the handling of optional arguments to subprograms, whereas the popular Fortran 77 extension `NUMARG()` was error-prone. The combination of all these features in Fortran 90 increase the efficiency of the resulting executable by giving the compiler more information with which to optimize the code.